# ¿Qué es la arquitectura hexagonal?

La arquitectura hexagonal es un patrón de diseño de software que busca aislar la lógica de negocio (core) de los detalles externos como bases de datos, APIs, interfaces de usuario o frameworks.

Su propósito principal es permitir que el núcleo del sistema sea independiente, testeable y fácil de adaptar a diferentes entornos.

# Dominio (o core)

Es el corazón del sistema, donde vive la lógica del negocio puro, libre de dependencias externas.

Contiene:

Entidades

Value Objects

Reglas de negocio

# Aplicación

Contiene los casos de uso del sistema, orquesta la ejecución de reglas del dominio.

Contiene:

Casos de uso (CreateOrder, LoginUser)

Interfaces de puertos de salida (UserRepository, EmailSender)

Servicios de aplicación

# Infraestructura

Es la capa que conecta el sistema con el mundo exterior: BD, API REST, colas, etc.

Contiene:

Adaptadores de entrada (Controllers, CLI, GraphQL...)

Adaptadores de salida (repositorios, servicios externos)

Configuración de tecnologías (NestJS, Express, Spring...)

# ¿Qué es Event Sourcing(Persistencia de datos)?

Event Sourcing es un patrón de arquitectura donde el estado de una entidad (por ejemplo, un pedido) no se guarda directamente, sino que se reconstruye a partir de una secuencia de eventos que describen todos los cambios que ha sufrido.

Ventajas:

📜 Auditoría total: sabes exactamente qué ocurrió y cuándo.

🔁 Deshacer cambios: puedes "rebobinar" o "replayear" estados.

📈 Proyecciones para diferentes vistas (por ejemplo, informes).

💥 Integración con CQRS y event-driven architecture fácilmente.

# Patron CQRS(Command and query segregation)

Separar el modelo de lectura del modelo de escritura en una aplicación:

Commands (comandos): modifican el estado del sistema (crear, actualizar, eliminar).

Queries (consultas): solo leen información (sin cambiar nada).

Arquitectura de micro servicios

Event-Driven Architecture: Dispará un evento cuando hay cambios en la DB(Rabbit MQ)

RabbitMQ es un message broker (intermediario de mensajes) que permite que diferentes servicios se comuniquen de forma asíncrona enviando y recibiendo mensajes a través de colas.

# Patron SAGA

· Coordinar múltiples servicios que deben ejecutar operaciones en conjunto.

· Mantener consistencia eventual sin bloquear recursos.

· Revertir automáticamente cuando algo falla.

·

Arquitectura de micro servicios

sAGA orquestada: Gestiona el flujo enviando ordenes al los micro servicios

SAGA corogreafeada: Reacción mediante evento

# Nest js

NestJS es un framework backend moderno para Node.js construido con TypeScript y principios de programación orientada a objetos y funcional. Es ideal para construir microservicios escalables gracias a su arquitectura modular, soporte nativo para transportes asincrónicos (como Kafka, Redis, RabbitMQ, etc.) y patrones como CQRS, SAGA, y event-driven architecture.

# ¿Qué son los Subjects en RxJS?

Un Subject es una combinación de Observable y Observer:

Puedes emitir valores (next()).

Puedes suscribirte para recibir valores.

RxJS (Reactive Extensions for JavaScript) es una librería para programación reactiva basada en Observables.

Permite manejar eventos asíncronos, secuencias de datos, streams y transformaciones de forma declarativa, flexible y poderosa.

Son ideales cuando quieres que varias partes de tu aplicación compartan o reaccionen a un flujo común de datos.

| **Tipo** | **¿Reemite valores antiguos?** | **Valor inicial requerido** | **Comportamiento** |
| --- | --- | --- | --- |
| Subject | ❌ No | ❌ No | No guarda estado |
| BehaviorSubject | ✅ Último | ✅ Sí | Guarda y reemite el último valor |
| ReplaySubject | ✅ Todos (o N últimos) | ❌ No | Reemite valores anteriores a nuevos subs |
| AsyncSubject | ✅ Solo el último al complete() | ❌ No | Solo emite último valor cuando se completa |

# ¿Qué son los Streams nativos de Node.js?

Los streams en Node.js son interfaces para manejar datos de manera eficiente y asincrónica. En lugar de leer o escribir datos completos en memoria (como en archivos grandes), los streams permiten procesarlos por partes (chunks).

Descargar archivo 2gb y leerlo mientras descarga y procesarlo al mismo tiempo

## Tipos de Streams en Node.js

| **Tipo** | **Descripción** | **Método clave** |
| --- | --- | --- |
| Readable | Emiten datos para ser leídos | .read() / .on('data') |
| Writable | Reciben datos para ser escritos | .write() / .end() |
| Duplex | Son de lectura y escritura (como sockets) | Ambos |
| Transform | Como Duplex, pero además transforman datos | .transform() |

# Patron de diseño

| **Patrón** | **¿Para qué sirve?** | **Ejemplo práctico** |
| --- | --- | --- |
| **Singleton** | Asegura una sola instancia global | Logger, ConfigService |
| **Factory Method** | Crea objetos sin especificar clase concreta | FacturaFactory, NotificaciónFactory |
| **Abstract Factory** | Familia de objetos relacionados sin especificar clases | UI multiplataforma |
| **Builder** | Construcción paso a paso de objetos complejos | Constructores de DTO, XML o JSON |
| **Prototype** | Clonar objetos existentes | Clonado de plantillas o configuraciones |

# Principios solid

### ****Tabla de los Principios SOLID****

| **Principio** | **Nombre** | **Definición breve** | **Objetivo principal** | **Ejemplo práctico (resumen)** |
| --- | --- | --- | --- | --- |
| **S** | **Single Responsibility** | Una clase debe tener una sola razón para cambiar | Separar responsabilidades | InvoicePrinter imprime, InvoiceRepository guarda |
| **O** | **Open/Closed** | El código debe estar abierto a extensión pero cerrado a modificación | Extensibilidad sin modificar código - Polimorfismo si el area no sabemos a quien le vamos a calcular usando implement | Añadir nuevos descuentos sin tocar la clase principal |
| **L** | **Liskov Substitution** | Las subclases deben poder sustituir a su clase base sin romper el sistema | Herencia lógica y coherente | Ostrich no debería extender Bird si no puede volar |
| **I** | **Interface Segregation** | No forzar a implementar métodos que no se necesitan | Interfaces pequeñas y específicas | Separar Workable y Eatable en lugar de una sola interfaz |
| **D** | **Dependency Inversion** | Depender de abstracciones, no de implementaciones concretas | Bajo acoplamiento entre módulos | EmailService inyectado como NotificationService |

# ¿Qué es sanitizar entradas?

Sanitizar entradas significa limpiar y validar los datos que provienen de fuentes externas (como formularios, URL, headers, body de peticiones, etc.) para:

Eliminar datos maliciosos.

Prevenir ataques (XSS, SQL Injection, NoSQL Injection, etc.).

Garantizar que los datos sean seguros y válidos antes de procesarlos o almacenarlos.

### Node.js (con Express)

js

const express = require('express');const xss = require('xss-clean');const mongoSanitize = require('express-mongo-sanitize');

const app = express();

app.use(express.json());

app.use(xss()); // Protege contra XSS

app.use(mongoSanitize()); // Protege contra NoSQL injection

### NestJS (con Pipes y class-validator)

ts

import { IsEmail, IsString } from 'class-validator';

export class CreateUserDto {

@IsEmail()

email: string;

@IsString()

name: string;

}

# ¿Qué son las pruebas unitarias en Angular con Jasmine y Karma?

Las pruebas unitarias son un tipo de prueba de software que verifica que una unidad de código (por ejemplo, una función, método o componente) funcione de manera aislada y como se espera.

En Angular, se usan por defecto:

🧼 Jasmine: Framework de testing (describe, it, expect…).

🧪 Karma: Test runner que ejecuta las pruebas en navegadores reales.

# SonarQube

SonarQube (a veces llamado simplemente Sonar) es una herramienta de análisis estático de código fuente que permite detectar:

🐛 Bugs

🚨 Vulnerabilidades

🧹 Code smells (malas prácticas)

📉 Problemas de cobertura de pruebas unitarias

🔐 Problemas de seguridad (OWASP, CWE, etc.

# Single Threaded Event Loop

El Single Threaded Event Loop es el mecanismo central de ejecución en Node.js, que permite manejar múltiples operaciones concurrentes (como I/O, timers, HTTP, etc.) usando un solo hilo (thread) de ejecución principal.

# Observables

Los Observables son una abstracción clave de programación reactiva. Representan una fuente de datos asíncronos que pueden emitir múltiples valores a lo largo del tiempo.

Son la base de librerías como RxJS (usada en Angular, NestJS, etc.) y ofrecen una forma elegante de trabajar con eventos, streams, APIs, WebSockets, etc.

| **Abstracción** | **Emite múltiples** | **Es asíncrono** | **Cancelable** | **Pull vs Push** |
| --- | --- | --- | --- | --- |
| Promise | ❌ una vez | ✅ | ❌ | Pull |
| Observable | ✅ múltiples | ✅ | ✅ | Push |

# Callbacks

Una función que se pasa como argumento a otra función, y se ejecuta más tarde (cuando termine la tarea asíncrona).

# Promises

Una promesa es un objeto que representa el resultado futuro (éxito o error) de una operación asíncrona.

# ASYNC / AWAIT

| **Palabra clave** | **¿Qué hace?** |
| --- | --- |
| async | Convierte una función en **asíncrona**, que **devuelve una Promise** |
| await | Espera el resultado de una Promise **antes de continuar** la ejecución |

# ¿Qué son PostgreSQL y MySQL?

Tanto PostgreSQL como MySQL son sistemas de gestión de bases de datos relacionales (RDBMS) que permiten:

Almacenar y consultar datos estructurados en tablas.

Usar SQL (Structured Query Language) para interactuar con los datos.

Aplicar reglas de integridad, relaciones entre tablas (FOREIGN KEY), índices, transacciones, etc.

| **Elemento** | **PostgreSQL** | **MySQL** |
| --- | --- | --- |
| Tipo | RDBMS avanzado y extensible | RDBMS simple y rápido |
| Mejor en... | Complejidad, integridad, JSON | Lecturas simples, rendimiento web |
| SQL estándar | Muy estricto | Más flexible con errores |
| Escalabilidad | Vertical y horizontal | Muy buena lectura vertical |

# ¿Qué es una base de datos no relacional?

Una base de datos no relacional (también llamada NoSQL) es un sistema de almacenamiento de datos que no utiliza el modelo tabular tradicional (filas y columnas), sino estructuras más flexibles, como:

Documentos

Claves y valores

Columnas

Grafos

Se usa especialmente cuando se necesita escalabilidad, flexibilidad de esquemas o altas velocidades de lectura/escritura.

# ¿Qué es un índice en una base de datos?

Un índice es una estructura de datos que la base de datos usa para acelerar las consultas sobre una tabla o colección.

Es similar al índice de un libro: te permite encontrar rápidamente la página donde está la información sin tener que leer todo el contenido.

# ¿Qué es un Pipe en Angular?

Un Pipe en Angular es una función pura y reutilizable que transforma los datos antes de mostrarlos en la vista (HTML).